Sketch-Guided GUI Test Generation for Mobile Applications

Chucheng Zhang, Haoliang Cheng, Enyi Tang, Xin Chen, Lei Bu, Xuandong Li

Automated Software Engineering 2017

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.
Sketch-Guided GUI Test Generation for Mobile Applications

Chucheng Zhang Haoliang Cheng Enyi Tang∗ Xin Chen Lei Bu Xuandong Li
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Software Institute of Nanjing University, Nanjing University, Nanjing, China
∗Corresponding author: eytang@nju.edu.cn

Abstract—Mobile applications with complex GUIs are very popular today. However, generating test cases for these applications is often tedious professional work. On the one hand, manually designing and writing elaborate GUI scripts requires expertise. On the other hand, generating GUI scripts with record and playback techniques usually depends on repetitive work that testers need to interact with the application over and over again, because only one path is recorded in an execution. Automatic GUI testing focuses on exploring combinations of GUI events. As the number of combinations is huge, it is still necessary to introduce a test interface for testers to reduce its search space.

This paper presents a sketch-guided GUI test generation approach for testing mobile applications, which provides a simple but expressive interface for testers to specify their testing purposes. Testers just need to draw a few simple strokes on the screenshots. Then our approach translates the strokes to a testing model and initiates a model-based automatic GUI testing.

We evaluate our sketch-guided approach on a few real-world Android applications collected from the literature. The results show that our approach can achieve higher coverage than existing automatic GUI testing techniques with just 10-minute sketching for an application.

I. INTRODUCTION

Until November 2016, the number of available mobile applications in the Google Play store is over 2.5 million [4]. Many of these applications provide rich features that help billions of users to communicate with each other. Hence, testing these applications is important. However, as most of the mobile applications interact with users through a graphical user interface (GUI), generating test cases for these GUI applications is a challenge.

Two classical approaches have been widely applied to GUI testing: manually writing GUI scripts as test cases that describe sequences of GUI actions, or recording the sequences of GUI events as test cases for playback later when testers execute the application under test. Both approaches strongly rely on manual work of professional testers. On the one hand, testers need to design and write quite a number of elaborate GUI scripts with their expertise in GUI testing. On the other hand, record and playback sequences of GUI events often depend on repetitive work that testers need to interact with the application over and over again, because only one path is recorded in an execution.

To free testers from the burden of tedious manual work, researchers have proposed a few techniques that generate GUI test cases automatically [3, 5, 6]. These techniques generate sequences of GUI events essentially by exploring the possible event combinations in applications. As the number of combinations of GUI events is often huge, how to reduce the exploring space (search space) for the automatic testing techniques is an important problem.

In this paper, we propose a sketch-guided GUI test generation approach for testing mobile applications, which covers combinations of GUI events following a sketch-based specification from testers. Testers just draw a few simple strokes on the screenshots as the sketches to specify their testing purposes. Then our approach translates the sketches to a testing model and initiates a model-based automatic GUI testing.

For example, when testers want to explore the GUI events effectively in the game Angry Birds1 (shown in Figure 1), they need to generate test cases that shoot a bird at the slingshot every time by dragging it as shown in Figure 1a. With different shooting angles and power, they get different scores. Existing techniques are difficult to uncover the high score GUI test case in the game, which is hidden in a lot of GUI event sequences

1http://www.angrybirds.com
that dragging the bird in different ways. The manual techniques need tedious labor effort to replay the shooting over and over again with various shooting angles and power, while other automatic techniques try to traverse all the GUI components in the application other than just dragging the bird in different ways.

Our sketch-guided approach is good at working on such testing tasks. Testers just simply draw a few strokes as Figure 1b, which specifies their testing purposes. In other words, they specify a range of GUI event sequences that they want our test engine to explore deeply by a simple sketch. The sketch in Figure 1b consists of 3 stroke components with different colors: a dragging action in blue, a range specifier in purple, and an existential quantifier in red. The dragging stroke makes our test engine focus on the bird-shooting GUI inputs during the testing, and the range specifier specifies the range to which the generated test scripts should drag the bird. At last, testers draw a quantifier to specify the target of the testing, which is just to find one GUI event sequence that get the score higher than a value in the game.

II. SKETCH-GUIDED APPROACH

This section presents the technical details of our sketch-guided GUI test generation. Testers provide a subject program for testing, and draw a few input sketches on the screenshots taken by our testing framework. Our framework processes the sketch with 2 stages, and outputs GUI test scripts accordingly. In the first stage, the framework recognizes the input sketches to a symbolic layout with attributes ripping from the subject program. Then in the second stage, it builds a model from the symbolic layout defined by the input sketches, and generates GUI test cases by traversing paths in the model. The rest of this section describes the technical details of every stage separately.

A. Sketching Language & Sketch Recognition

The first stage of our workflow recognizes primitive shapes in the input sketches, and generates a symbolic sketch layout that holds the information of shapes and matches the GUI widgets to each corresponding shape.

We design an expressive but simple sketching language, which defines only 4 types of primitive shapes: the action strokes, range specifiers, quantifiers, and boolean connectors. Figure 2-5 present the primitive shapes in our sketching system. Every action stroke describes a user action in the GUI testing, such as touching on a button or dragging an icon. After drawing an action stroke, the tester draws immediately a range specifier to delineate a set of GUI components that are affected by the action. Quantifiers and boolean connectors further specify the logical conditions and logical relations of the actions with different GUI components, so we also call them logical shapes. For example, the tester can draw an existential quantifier (∃) after a range specifier, which means the testing system just needs to find one GUI component in the specified range that makes the event sequence satisfy the test oracle.

Note that not everything in our framework is represented by the sketch. Specifically, testers specify the text-input actions and the test oracle directly through a textbox in our system and links them to the symbolic layout later after other sketches are recognized. The test oracle is some constraints that should be satisfied when the generated test cases are performed, such as Color(x,y)==RGB(200,0,0) where x and y are coordinates of a user-specified point. If the application crashes during testing, the test oracle is always unsatisfied.
Figure 2 shows several typical types of action strokes, which specify the user actions in the application’s GUI systems. For a touchscreen-based mobile application, the typical actions such as touching, long touching a GUI widget, or dragging a widget to another place are expressed separately as strokes in Figure 2a, Figure 2b, and Figure 2c. Testers can further extend the system with more GUI actions when necessary. Our sketching system not only recognizes the action strokes, but also records the coordinates of key points and its corresponding GUI widgets specified by the strokes. For example, when a tester draws a drag action on the screenshot, our system records the coordinates of its start point and end point along with the information of GUI widgets at these points.

Our system rips the information of GUI widgets from the application under test when we take the screenshots, and binds every screenshot with the bound of widgets on it. When the tester draws an action on a screenshot, our system directly obtains the information of widgets from the binding. The range specifiers shown in Figure 3 also use such a technique to collect the widget information in the range. Usually, the range specifiers denote a set of GUI widgets in the application as Figure 3a and Figure 3b show. In some Games, it also denotes a set of coordinates on the screen when our system does not get any widgets in the range like Figure 1b.

For every range specifier, we designate a quantifier in our sketching language. The quantifier specifies the quantity of events in the range that satisfy the oracle, which affects the search condition in our test generation. For example, the universal quantifier (\(\forall\)) in Figure 4b requires all events in the range to eventually satisfy the condition in the test oracle, whereas the existential quantifier (\(\exists\)) in Figure 4a drives the test engine to search for just one event that satisfies the condition, which is often used in generating a GUI event that triggers the bug. We introduce 2 recursive quantifiers in our sketching language shown in Figure 4c and Figure 4d. We call them the recursive existential quantifier (\(\exists\)) and the recursive universal quantifier (\(\forall\)) separately, which recursively specify the quantity of all subsequent actions of the current event in the range.

We also introduce the boolean connectors in our sketching language to make it expressive and powerful. Figure 5 presents the boolean connectors in our system, which represents boolean and (\(\&\)) in Figure 5a, or (\(\|\)) in Figure 5b, and not (\(\neg\)) in Figure 5c.

The following context-free grammar (also called BNF, Backus-Naur form) defines the syntax of our sketching language:

\[
\text{sketch} \rightarrow \text{tracelist} \\
\text{tracelist} \rightarrow \text{eventtrace} \mid \text{tracelist, eventtrace} \\
\text{eventtrace} \rightarrow \text{eventstep} \mid \text{eventstep}; \text{eventstep} \\
\text{eventstep} \rightarrow \text{actionset} \mid \text{eventstep boolean actionset} \\
\text{actionset} \rightarrow \text{act} \mid \text{act + range + quant}
\]

where the act, range, quant, and boolean in the grammar means an action stroke, range specifier, quantifier, and boolean connector separately, and the punctuation marks in the grammar such as +, .; and , are gestures or buttons in our sketching system that specify the relations of shapes in the sketch. The grammar defines the drawing order for testers in producing their sketches. Testers start their sketching by drawing an action stroke to specify an event (such as touch) on a GUI widget. Then they optionally add a range specifier to make the event affect on a set of widgets, and a quantifier to present the quantity of events in the range that satisfy the oracle. At this point, testers have specified an actionset with actions on different widgets, and they draw boolean connectors to connect multiple actionsets and build an eventstep in their testing. An eventtrace is either a sequence of eventsteps connected with semicolons in the grammar, or a fork from an eventstep in the trace to a list of eventtraces with a colon at the fork point. A sketch is finally a tracelist that consists of multiple eventtraces.

### B. Event-flow Modeling & Script Generation

The second stage of our workflow builds a GUI model for test generation from the symbolic sketch layout generated by the previous stage. The symbolic sketch layout \(L\) is defined as a 3-tuple \((l, W, O)\), where \(l\) is a symbolic tracelist organized as the grammar in Equation 1, \(W\) is a mapping that maps the coordinates in \(l\) to the widgets in the application under test, and \(O\) is another mapping that maps every eventtrace in \(l\) to a test oracle specified by the tester.

The coordinate-widget map \(L \rightarrow W\) in the sketch layout \(L\) stores the bound of every widget in the application under test. When our framework queries the coordinates of a point \(p\), on a screenshot, the map \(L \rightarrow W(p)\) analyzes the bound of every widget and returns the widget that the sketch locates the point in. We translate all coordinates to widgets in our GUI model. So the output test scripts can directly perform actions on the GUI widgets, which generalizes the testing on devices with different screen resolutions.

Our framework models the sketch layout as a partial abstract event-flow graph (P-aEFG), which is a variation of the event-flow graph (EFG) defined by Memon et al. for GUI testing [13]. Not as the EFG models representing all possible event sequences on a GUI, our P-aEFG just partially extracts the abstract event-flow that represents possible interactions defined by the sketch. Hence, the P-aEFG is often smaller. In our P-aEFG, a vertex represents an abstract event that summarizes the events may be performed at the same application state. And an edge in a P-aEFG from the vertex \(v_1\) to \(v_2\) means that an event \(e_2\) in the abstract event \(v_2\) may be performed immediately after every event \(e_1\) in the abstract event of \(v_1\), and every sequence of \(e_1, e_2\) should be tested when it is specified by the sketch. We formally define a P-aEFG (\(G\)) as a 4-tuple \((V, E, V_0, O)\), where:

1) \(V\) is a vertex set and each vertex \(v \in V\) is an abstract event specified in the sketch. An abstract event represents a set of events that may be performed at the same application state.
Algorithm 1 Abstract Event-flow Modeling

Input: \( L(l, W, O) \)

Output: \( G(V, E, V_0, O) \)

1. for all \( es \in \text{eventStep}(L) \) do
2. \( t_v \leftarrow \text{syntaxTree}(es) \)
3. for all \( p_t \in \text{pointCoordinate}(t_v) \) do
4. \( t_v \leftarrow t_v[p_t \rightarrow L.W(p_t)] \) \# substitute coordinates to widget
5. end for
6. \( V(es) \leftarrow t_v \) \# hold the vertex for every eventStep
7. \( G.V \leftarrow G.V \cup \{t_v\} \)
8. end for
9. for all “\( es_1; es_2 \)” \( \in \text{substring}(L) \) do
10. \( G.E \leftarrow G.E \cup \{(V(es_1), V(es_2))\} \)
11. end for
12. for all “\( \text{et} : \text{tlist} \)” \( \in \text{substring}(L) \) \& \( \forall \text{et}' \in \text{tlist} \) do
13. \( es_1 \leftarrow \text{lastEventStep(et)} \)
14. \( es_2 \leftarrow \text{firstEventStep(et')’} \)
15. \( G.E \leftarrow G.E \cup \{(V(es_1), V(es_2))\} \)
16. end for
17. for all \( \text{et} \in \text{eventTrace}(L) \) do
18. \( es \leftarrow \text{firstEventStep(et)} \)
19. \( G.V_0 \leftarrow G.V_0 \cup \{V(es)\} \)
20. end for
21. \( G.O \leftarrow \text{oracleParse}(L, V) \)

Algorithm 2 oracleParse: Marks Every Test Oracle on the Path of the Abstract Event-flow Graph

Input: \( L(l, W, O), V \) \# the map holds the vertex for each eventStep

Output: \( O \) \# the map binds every oracle to the path

1. for all \( \text{et} \in \text{eventTrace}(L) \) do
2. \( P \leftarrow P \cup \{p\} \) \# put the path \( p \) to the set \( P \)
3. \( \forall p \in P \) do
4. if “\( es_1; es_2 \)” \( \in \text{substring(et)} \)
5. \( \land V(es_1) \implies \text{lastVertex}(p) \) then
6. \( p' \leftarrow \text{addVertex}(p, V(es_2)) \) \# catenate a vertex at the end
7. else if “\( \text{et} : \text{tlist} \)” \( \in \text{substring(et)} \)
8. \( \land V(\text{lastEventStep(et'}) \) \( \implies \text{lastVertex}(p) \) then
9. \( P \leftarrow P \setminus \{p\} \)
10. for all \( \text{et''} \in \text{tlist} \) do
11. \( p' \leftarrow \text{addVertex}(p, V(\text{firstEventStep(et'')}) \)
12. \( P \leftarrow P \cup \{p'\} \)
13. end for
14. end if
15. \( \forall p \in P, O(p) \leftarrow L.O(p) \)
16. \( P \leftarrow \varnothing \)
17. end for

2) \( E \subseteq V \times V \) is a directed edge set between vertices. An edge \( (v_1, v_2) \in E \) iff \( \forall e_1 \in v_1, \forall e_2 \in v_2, e_1 \) may be performed immediately after \( e_2 \), and the combination of \( e_1, e_2 \) should be tested according to the specification.

3) \( V_0 \) is a set of start vertices representing the events that are available for the testers when the application starts.

4) \( O \) is a map that binds every test oracle to the path in the graph. A path \( p \) in the P-aEFG is a sequence of vertices \( (v_1, v_2, \ldots, v_n) \in V \times V \times \ldots \times V \), while a test oracle \( o \) is a constraint that the application should satisfy when the corresponding sequence of events have been performed.

Algorithm 1 parses the sketch layout \( L \) and builds the P-aEFG model \( G \) for test case generation. It first generates the vertices of the graph from the sketch layout (line 1-8). The function eventStep at line 1 returns a set of all top-level eventSteps that is syntactically not a substring of any other eventStep in the sketch layout. As every eventStep denotes the events may be performed at the same application state, we generate the vertex of our P-aEFG by substituting the coordinates of points to the corresponding widgets in the syntax tree of an eventStep. After the substitution, every range specifier in the symbolic sketch layout becomes a set of corresponding GUI widgets in the abstract event. Line 9-16 of Algorithm 1 links vertices with edges depending on the gestures (a semicolon or a colon) in the sketch layout. Then it stores all the start vertices in \( G.V_0 \) (line 17-20). Finally, the algorithm marks every test oracle on the path of the P-aEFG by calling oracleParse at line 21, and stores it in the map \( G.O \) of the P-aEFG.

Algorithm 2 depicts details of the function oracleParse, which extracts the oracle constraints from the symbolic layout \( L.O \), and marks them on the path \( p \) in the P-aEFG. Since testers have specified the oracle constraints for every eventtrace during sketching, Algorithm 2 seeks the set of paths \( P \) that corresponds to the symbolic event trace \( \text{et} \) (line 4-14), and marks the oracle constraints in the returned map (line 15).

After building the P-aEFG model \( G \), our framework generates test cases directly by traversing the P-aEFG model. Generally, every GUI test case corresponds to a path in \( G \), which is defined by a sequence of GUI events and an oracle constraint in our testing system. For every path \( p \) in the model \( G \), our framework picks the first event from the start set \( G.V_0 \), develops a sequence of GUI events by picking the events one by one along the path, gets the oracle constraint from \( G.O(p) \), and outputs the test case to a GUI test script.

III. Evaluation

We have implemented a prototype of our sketch-guided testing system\(^2\), and performed our evaluation on 10 Android applications collected by Choi et al. [9], which are listed in Table I. All the applications are open source projects from F-Droid app market. The smallest project is music note with 1345 instructions in its bytecode, whereas the largest one is anymemo with 72145 bytecode instructions. Further details of each project can be referred to [9, 10].

We recruited 12 volunteers who were 3rd-year college students in Computer Science from our university to evaluate the system. All of them are familiar with Android devices but only 6 of them have taken the software testing class in our university. We pair every one of them with a student who has

\(^2\)http://software.nju.edu.cn/eytang/artifacts/sketch/sketchgeneration.tar.gz.
We perform every tool on the same platform and environment with SwiftHand.

Emma provided by the Android development kit, which creates random inputs without considering application's state. An-

testers cannot specify meaningful guidance other than randomly surfing in the file system. So the line coverage of Monkey, which creates random inputs to test the application, is similar to our sketch-guided testing on explorer.

However, SwiftHand applies a machine learning strategy to the control flow, which focuses on the internal structure of the application and achieves higher coverage on explorer. Nonetheless, our sketch-guided testing achieves higher branch coverage than SwiftHand on most applications, which indicates that testers' guidance is still better than machine learning algorithms with current technology.

From the results in Table I, all the 4 GUI testing techniques achieve low coverage on a few applications such as sanity, because a large number of modules and features in the application are not activated by GUI actions, but the environments, such as sensors, GPS, etc. Our sketch-guided approach does not support the environment testing currently. Such limitation can be overcome by extending our sketching system in future.

We collect the sketches at different scheduled time from testers in our testing, and compute the code coverage from the sketches of different sketching time to estimate the necessary manual effort in our sketch-guided testing. Table II presents the branch coverage that our sketch-guided testing achieves with different sketching time. From Table II, simple mobile applications only need a very short sketching time (just less than 2-4 minutes) to achieve a stable code coverage, and all the applications in our evaluation can achieve good coverage in about 8-minute sketching time. Hence, the manual effort in our sketch-guided testing is acceptable.

IV. RELATED WORK

As smartphones and tablets become very popular today, various techniques focus on testing GUIs of mobile applications, such as guiding random testing with different strategies [11],

<table>
<thead>
<tr>
<th>App</th>
<th>% Line Coverage</th>
<th>% Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Monkey</td>
<td>AndroidRipper</td>
</tr>
<tr>
<td>music note</td>
<td>46.85</td>
<td>4.01</td>
</tr>
<tr>
<td>whohas</td>
<td>59.33</td>
<td>16.31</td>
</tr>
<tr>
<td>explorer</td>
<td>69.36</td>
<td>2.52</td>
</tr>
<tr>
<td>weight</td>
<td>45.07</td>
<td>14.97</td>
</tr>
<tr>
<td>tipsy</td>
<td>78.62</td>
<td>7.08</td>
</tr>
<tr>
<td>myexpense</td>
<td>42.57</td>
<td>12.31</td>
</tr>
<tr>
<td>mininote</td>
<td>29.43</td>
<td>4.25</td>
</tr>
<tr>
<td>mileage</td>
<td>30.60</td>
<td>5.52</td>
</tr>
<tr>
<td>anymemo</td>
<td>25.51</td>
<td>3.99</td>
</tr>
<tr>
<td>sanity</td>
<td>22.17</td>
<td>3.99</td>
</tr>
<tr>
<td>Average</td>
<td>44.93</td>
<td>4.89</td>
</tr>
</tbody>
</table>

TABLE I
CODE COVERAGE COMPARISON OF THE SKETCH-GUIDED TESTING AND A FEW RECENT AUTOMATIC GUI TESTING TECHNIQUES

<table>
<thead>
<tr>
<th>App</th>
<th>2 min</th>
<th>4 min</th>
<th>6 min</th>
<th>7 min</th>
<th>8 min</th>
<th>after 2 minute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extra Feedback</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>music note</td>
<td>52.75</td>
<td>72.73</td>
<td>73.31</td>
<td>73.61</td>
<td>73.63</td>
<td>73.63</td>
</tr>
<tr>
<td>whohas</td>
<td>25.32</td>
<td>60.83</td>
<td>61.28</td>
<td>61.36</td>
<td>61.39</td>
<td>61.39</td>
</tr>
<tr>
<td>explorer</td>
<td>45.39</td>
<td>47.30</td>
<td>47.31</td>
<td>47.36</td>
<td>47.37</td>
<td>53.42</td>
</tr>
<tr>
<td>weight</td>
<td>42.40</td>
<td>42.45</td>
<td>42.48</td>
<td>42.48</td>
<td>42.48</td>
<td>62.02</td>
</tr>
<tr>
<td>tipsy</td>
<td>16.84</td>
<td>41.44</td>
<td>44.30</td>
<td>44.36</td>
<td>44.39</td>
<td>65.82</td>
</tr>
<tr>
<td>myexpense</td>
<td>6.04</td>
<td>9.83</td>
<td>18.08</td>
<td>36.45</td>
<td>38.64</td>
<td>45.42</td>
</tr>
<tr>
<td>mininote</td>
<td>25.49</td>
<td>25.68</td>
<td>25.77</td>
<td>25.77</td>
<td>25.80</td>
<td>28.14</td>
</tr>
<tr>
<td>mileage</td>
<td>16.83</td>
<td>20.07</td>
<td>25.68</td>
<td>32.37</td>
<td>37.05</td>
<td>39.64</td>
</tr>
<tr>
<td>anymemo</td>
<td>4.95</td>
<td>17.18</td>
<td>23.57</td>
<td>29.79</td>
<td>33.31</td>
<td>41.67</td>
</tr>
<tr>
<td>sanity</td>
<td>2.17</td>
<td>5.39</td>
<td>9.96</td>
<td>11.45</td>
<td>11.68</td>
<td>18.69</td>
</tr>
</tbody>
</table>

TABLE II
BRANCH COVERAGE (% OF THE SKETCH-GUIDED TESTING WITH DIFFERENT SKETCHING TIME

3http://emma.sourceforge.net/
The results show that our approach can achieve higher coverage for testing mobile applications. Testers just need to draw a few simple strokes on the screenshots following the syntax of the sketching language we defined in our system. Then our approach translates the strokes to the Partial Abstract Event-Flow Graph (P-aEFG), and initiates a model-based automatic GUI testing. We evaluate our sketch-guided approach on a few real-world Android applications collected from the literature. The results show that our approach can achieve higher coverage than existing automatic GUI testing techniques with just 10-minute sketching for an application. In the future, we will extend our approach to support in testing the environment of mobile applications, which guides the application in different device environments and uncovers more bugs consequentially.

V. CONCLUSION AND FUTURE WORK

We present a sketch-guided GUI test generation approach for testing mobile applications. Testers just need to draw a few simple strokes on the screenshots following the syntax of the sketching language we defined in our system. Then our approach translates the strokes to the Partial Abstract Event-Flow Graph (P-aEFG), and initiates a model-based automatic GUI testing. We evaluate our sketch-guided approach on a few real-world Android applications collected from the literature. The results show that our approach can achieve higher coverage than existing automatic GUI testing techniques with just 10-minute sketching for an application. In the future, we will extend our approach to support in testing the environment of mobile applications, which guides the application in different device environments and uncovers more bugs consequentially.

ACKNOWLEDGEMENTS

This research was supported in part by National Natural Science Foundation of China (No. 61772260, 61402222, 61632015 and 61690204) and National Key Research and Development Program of China (No. 2016YFB1000802).

REFERENCES